



**AMERICAN
UNIVERSITY_{OF} BEIRUT**

**MAROUN SEMAAN FACULTY OF
ENGINEERING & ARCHITECTURE**

EECE 435L - Project

**Hashem Khodor & Mahmoud Yamani
FALL 2024/2025**

**Report Submission Date: Dec. 03, 2024
American University of Beirut**

Contents

Chapter 1	Introduction	5
1.1	Project Overview	5
1.2	Objectives	5
Chapter 2	System Architecture	6
2.1	Service Description	6
2.1.1	Customers Service	6
2.1.2	Inventory Service	6
2.1.3	Sales Service	6
2.1.4	Reviews Service	6
Chapter 3	Implementation Details	7
3.1	Service-Specific Implementation	7
3.1.1	Service 1 - Customers	7
3.1.2	Service 2 - Inventory	7
3.1.3	Service 3 - Sales	8
3.1.4	Service 4 - Reviews	9
3.2	API Documentation	9
3.2.1	Customer Service	9
3.2.2	Inventory Service	16
3.2.3	Sales Service	20
3.2.4	Reviews Service	23
3.3	Database Schema Design	29
3.3.1	ENUM Types	29
3.3.2	Tables	29
3.3.3	Relationships	29
3.4	Database Schema Diagram	30
3.5	SQL Schema Definition	30
Chapter 4	Error Handling and Validation	34
4.1	Error Management	34
4.2	Validation	34
Chapter 5	Testing	35
5.1	Overview of Tools and Testing Strategy	35
5.2	Tools and Libraries	35
5.3	Customer Service	36
5.3.1	Unit Tests	36
5.3.2	Integration Tests	36

5.3.3	Validation and Error Handling Tests	36
5.3.4	Edge Case Tests	37
5.4	Review Router	37
5.4.1	Unit Tests	37
5.4.2	Integration Tests	38
5.4.3	Validation and Error Handling Tests	38
5.4.4	Edge Case Tests	38
5.4.5	Moderation Workflow Tests	38
5.5	Inventory Service	39
5.5.1	Unit Tests	39
5.5.2	Integration Tests	39
5.5.3	Validation and Error Handling Tests	40
5.5.4	Edge Case Tests	40
5.5.5	Business Logic Tests	40
5.5.6	Performance and Scalability Tests	40
5.6	Sales Service	40
5.6.1	Unit Tests	41
5.6.2	Integration Tests	41
5.6.3	Validation and Error Handling Tests	41
5.6.4	Edge Case Tests	42
5.6.5	Workflow Tests	42
5.6.6	Performance and Scalability Tests	42
Chapter 6	Deployment and Integration	43
6.1	Containerization of Services	43
6.1.1	Dockerfiles for Each Service	43
6.1.2	Service-Specific Configurations	43
6.1.3	Service Orchestration with Docker Compose	44
Chapter 7	Documentation and Profiling	45
7.1	Documentation	45
7.2	Performance Profiling	45
Chapter 8	GitHub and Version Control	51
8.1	Repository Link	51
Chapter 9	Validation and Sanitization	52
9.1	Use of Pydantic for Input Validation	52
9.2	Parameterization and Structured Queries	53
9.3	Sanitization and Data Integrity Checks	53
Chapter 10	User Authentication	55
10.1	Token-Based Authentication	55
10.2	Authorization Management	55
10.2.1	Role-Based Access Control (RBAC)	55
Chapter 11	Moderation	56
11.1	Moderation Features for Inappropriate Reviews	56
11.1.1	Moderation Process Implementation	56

11.1.2 JWT Implementation for Authentication	56
Chapter 12 Additional Professional Tasks	58
12.1 JWT-Based Authentication	58
12.2 Logging and Error Monitoring	58
12.3 Asynchronous Operations	58
12.4 Cloud-Hosted Database with Indexing	59
12.5 Health Checks	60
References	61
Chapter A Appendix A: Supporting Materials	62

List of Figures

3.1 Database Schema Diagram	32
5.1 Output of pytest showing test results for the Customer module.	37
5.2 Output of pytest showing test results for the Review module.	39
5.3 Output of pytest showing test results for the Inventory Service module.	41
5.4 Output of pytest showing test results for the Sales Service module.	42
7.1 HTML Documentation	45
7.2 Sample profiling output showing memory usage of customer service.	46
7.3 Sample profiling output showing execution time of customer service.	46
7.4 Customer service code coverage.	47
7.5 Sample profiling output showing memory usage of inventory service.	47
7.6 Sample profiling output showing execution time of inventory service.	47
7.7 Inventory service code coverage.	48
7.8 Sample profiling output showing memory usage of review service.	48
7.9 Sample profiling output showing execution time of review service.	49
7.10 Review service code coverage.	49
7.11 Sample profiling output showing memory usage and execution time of sales service.	50
7.12 Sales service code coverage.	50
9.1 Example of Parametrized and Structured Query using supabase client	53
9.2 Example Pydantic model with field validations	54
12.1 Sample logging output showing execution details with timestamp.	58
A.1 Customer Charge Wallet - 200 Response	62
A.2 Customer Charge Wallet - 404 Response	62
A.3 Customer Deduct - 200 Response	62
A.4 Customer Deduct - 404 Response	63

A.5	Customer Delete - 200 Response	63
A.6	Customer Get - 200 Response	63
A.7	Customer Get All Response	63
A.8	Customer Register - 200 Response	64
A.9	Customer Register - 404 Response	64
A.10	Customer Register - 409 Response	64
A.11	Customer Update - 200 Response	65
A.12	Customer Update - 404 Response	65
A.13	Good Add - 200 Response	65
A.14	Good Deduct - 200 Response	65
A.15	Good Get - 200 Response	66
A.16	Good Update - 200 Response	66
A.17	Purchase - 200 Response (1)	66
A.18	Purchase - 200 Response	66
A.19	Sales Get - 200 Response	67
A.20	Postman - Test 1	68
A.21	Postman - Test 2	69
A.22	Postman - Final Test	70

List of Tables

3.1	Customer Table	30
3.2	Wallet Table	30
3.3	Inventory Table	30
3.4	Review Table	31
3.5	Purchases Table	31

1. Introduction

1.1 Project Overview

This project involves the development of an E-commerce platform that facilitates online shopping for customers. The system encompasses various services for Customer Management, Inventory Control, Sales Processing, and Review Handling. The purpose of the project is to create a platform that streamlines backend operations.

1.2 Objectives

- Customers can browse and purchase products.
- Implement robust inventory management to track product stock levels.
- Create a sales processing system to handle transactions.
- Incorporate a review system for customers to provide feedback.

2. System Architecture

2.1 Service Description

The ecommerce system is divided into four main services:

2.1.1 Customers Service

Handles user registration, authentication, profile management, and user data storage.

2.1.2 Inventory Service

Manages product listings, stock levels, product details, and categorization.

2.1.3 Sales Service

Processes orders, handles payments, maintains order history, and manages shopping carts.

2.1.4 Reviews Service

Allows customers to submit reviews, moderates content, and provides feedback mechanisms.

3. Implementation Details

3.1 Service-Specific Implementation

3.1.1 Service 1 - Customers

Functionality The **Customers Service** is responsible for all operations related to customer management. Its primary functionalities include:

- **Customer Registration:** Allows new users to register by providing personal details.
- **Authentication:** Handles user login requests and verifies credentials.
- **Profile Management:** Enables customers to update their personal information such as name, age, address, gender, and marital status.
- **Wallet Management:** Provides functionalities to charge and deduct funds from a customer's wallet, facilitating transactions within the platform.

Challenges Faced During the implementation of the Customers Service, several challenges were encountered:

- **Data Validation:** Ensuring the integrity and validity of customer data through rigorous validation checks.
- **Concurrency Control:** Managing concurrent wallet updates to prevent race conditions and ensure accurate wallet balances.
- **Error Handling:** Providing meaningful error messages and handling exceptions to improve the user experience and facilitate debugging.
- **Security:** Protecting sensitive customer information, especially passwords, through encryption and secure storage practices.

3.1.2 Service 2 - Inventory

Functionality The **Inventory Service** is responsible for managing all aspects related to the goods available on the e-commerce platform. Its primary functionalities include:

- **Adding New Products:** Allows administrators to add new goods to the inventory with details such as name, category, price, description, and stock count.

- **Updating Existing Products:** Enables updating information of existing goods, including price adjustments, description changes, and stock replenishment.
- **Retrieving Product Details:** Provides the ability to fetch information about specific goods using their unique identifiers.
- **Stock Deduction:** Manages the stock levels by deducting units when a sale occurs, ensuring accurate inventory tracking.

Challenges Faced During the development of the Inventory Service, the following challenges were encountered:

- **Concurrency Issues:** Ensure stock levels remain accurate.
- **Data Validation:** Ensuring that all inputs, such as price and count, are validated to prevent incorrect data from corrupting the inventory.
- **Error Handling:** Providing clear and informative error messages to aid in debugging and improve the user experience.
- **Database Performance:** Optimizing database queries and operations to handle large volumes of inventory data efficiently.

3.1.3 Service 3 - Sales

Functionality The **Sales Service** is responsible for processing sales transactions within the e-commerce platform. Its primary functionalities include:

- **Processing Purchases:** Manages the complete purchase process for goods, including verifying inventory availability, deducting stock, and updating customer wallet balances.
- **Recording Sales:** Maintains a record of all sales transactions for historical data analysis and reporting purposes.
- **Retrieving Purchase History:** Allows retrieval of all purchase records to facilitate auditing and customer service support.

Challenges Faced During the development of the Sales Service, several challenges were encountered:

- **Inter-Service Communication:** Ensuring reliable and secure communication between the Sales Service and other services like Inventory and Customer services.
- **Transaction Atomicity:** Maintaining transactional integrity to ensure that all steps in the purchase process either complete successfully or fail together, preventing inconsistencies.
- **Error Propagation:** Handling errors from dependent services gracefully and providing meaningful feedback to the client.
- **Concurrency Control:** Managing simultaneous purchase requests to prevent race conditions, such as overselling stock or overdrawing customer wallets.

3.1.4 Service 4 - Reviews

Functionality The **Reviews Service** is dedicated to managing customer reviews for items on the e-commerce platform. Its primary functionalities include:

- **Review Submission:** Allows customers to submit reviews for items they have purchased.
- **Review Updating:** Enables customers to update their existing reviews.
- **Review Deletion:** Allows customers to delete their reviews.
- **Review Moderation:** Provides moderators with the ability to approve, flag, or require approval for reviews.
- **Review Retrieval:** Facilitates the retrieval of reviews based on item IDs or customer IDs for display purposes.

Challenges Faced During the development of the Reviews Service, several challenges were encountered:

- **Authentication and Authorization:** Implementing secure authentication for moderators using JWT tokens and ensuring only authorized users can perform moderation actions.
- **Data Consistency:** Ensuring that reviews are correctly associated with existing customers and items, and handling cases where either may not exist.
- **Error Handling:** Providing informative error messages and handling exceptions to improve the user experience and facilitate debugging.
- **Moderation Workflow:** Designing a flexible moderation system that allows reviews to be flagged, approved, or set as needing approval.

3.2 API Documentation

3.2.1 Customer Service

POST /api/v1/customer/auth/register

Description: This endpoint registers a new customer in the system, creating both a customer record and an associated wallet.

Request Body Example:

```
{
  "name": "John Doe",
  "username": "johndoe",
  "password": "SecurePass123",
  "age": 30,
  "address": "123 Main St",
  "gender": true,
  "marital_status": "single"
}
```

Possible Responses:

- **201 Created**

```
{
  "message": "Registered 'johndoe' successfully",
  "data": {
    "username": "johndoe"
  }
}
```

- **400 Bad Request**

```
{
  "message": "Failed to register johndoe",
  "errors": "Invalid age provided"
}
```

- **409 Conflict**

```
{
  "message": "Customer with username 'johndoe' already exists",
  "errors": "Username already taken"
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try registering again later",
  "errors": "Database connection error"
}
```

DELETE /api/v1/customer/delete/{customer_id}

Description: Deletes an existing customer and their associated wallet by the given `customer_id` (username).

Possible Responses:

- **200 OK**

```
{
  "message": "Deleted 'johndoe' successfully"
}
```

- **400 Bad Request**

```
{
  "message": "Failed to delete customer with id 'johndoe'",
  "errors": "Unable to remove from database"
}
```

- 404 Not Found

```
{
  "message": "Customer with username 'johndoe' not found"
}
```

- 500 Internal Server Error

```
{
  "message": "Internal Server Error. Please try deleting again later",
  "errors": "Database timeout"
}
```

PUT /api/v1/customer/update/{customer_id}

Description: Updates customer information. Fields not provided remain unchanged. If no fields are provided, it returns a message indicating no change.

Request Body Example (partial update):

```
{
  "address": "456 Park Avenue",
  "age": 31
}
```

Possible Responses:

- 200 OK

```
{
  "message": "Updated 'johndoe' successfully",
  "data": {
    "username": "johndoe",
    "name": "John Doe",
    "age": 31,
    "address": "456 Park Avenue",
    "gender": true,
    "marital_status": "single",
    "role": "customer"
  }
}
```

- **202 Accepted**

```
{
  "message": "Customer update request for 'johndoe' processed,
             but no new data available"
}
```

- **404 Not Found**

```
{
  "message": "Customer with username 'johndoe' not found"
}
```

- **400 Bad Request**

```
{
  "message": "Failed to update 'johndoe'",
  "errors": "Invalid 'age' field"
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try updating again later",
  "errors": "Unknown server error"
}
```

GET /api/v1/customer/get

Description: Retrieves all customers currently stored in the system.

Possible Responses:

- **200 OK**

```
{
  "data":
  {
    "johndoe": {
      "username": "johndoe",
      "name": "John Doe",
      "age": 31,
      "address": "456 Park Avenue",
      "gender": true,
      "marital_status": "single",
      "role": "customer"
    }
  }
}
```

```
    },
    {
      "janedoe": {
        "username": "janedoe",
        "name": "Jane Doe",
        "age": 28,
        "address": "789 Oak Street",
        "gender": false,
        "marital_status": "married",
        "role": "customer"
      }
    }
  }
}
```

- 500 Internal Server Error

```
{
  "error": "Unable to retrieve customers from database"
}
```

GET /api/v1/customer/get/{customer_id}

Description: Retrieves a single customer's details along with their wallet information.

Possible Responses:

- 200 OK

```
{
  "message": "Retrieved customer 'johndoe' successfully",
  "data": {
    "user": {
      "username": "johndoe",
      "name": "John Doe",
      "age": 31,
      "address": "456 Park Avenue",
      "gender": true,
      "marital_status": "single",
      "role": "customer"
    },
    "wallet": {
      "customer_id": "johndoe",
      "amount": 100.0,
      "last_updated": "2024-12-08T12:34:56Z"
    }
  }
}
```

- **404 Not Found**

```
{
  "message": "Customer with username 'johndoe' not found"
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try retrieving again later",
  "errors": "Database query failed"
}
```

PUT /api/v1/customer/wallet/{customer_id}/charge

Description: Adds a specified non-negative amount to a customer's wallet.

Request Body Example:

```
{
  "amount": 50.0
}
```

Possible Responses:

- **200 OK**

```
{
  "message": "Wallet for customer 'johndoe' charged with 50.0",
  "data": {
    "new_balance": 150.0
  }
}
```

- **404 Not Found**

```
{
  "message": "Wallet for customer 'johndoe' not found"
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try charging again later",
  "errors": "Could not update the wallet balance"
}
```

PUT /api/v1/customer/wallet/{customer_id}/deduct

Description: Deducts a specified amount from a customer's wallet, if sufficient funds are available.

Request Body Example:

```
{
  "amount": 20.0
}
```

Possible Responses:

- **200 OK**

```
{
  "message": "20.0 deducted from wallet for customer 'johndoe'",
  "data": {
    "new_balance": 130.0
  }
}
```

- **400 Bad Request**

```
{
  "message": "Insufficient funds in wallet"
}
```

- **404 Not Found**

```
{
  "message": "Wallet or customer 'johndoe' not found"
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try deducting again later",
  "errors": "Unable to process the transaction"
}
```

GET /health

Description: Checks if the service and database are operational.

Possible Responses:

- **200 OK**


```
{
  "status": "OK",
  "db_status": "connected",
  "customers_count": 42
}
```

- **500 Internal Server Error** (or similar error status)

```
{
  "status": "ERROR",
  "db_status": "disconnected",
  "error": "Timeout connecting to database"
}
```

3.2.2 Inventory Service

The service offers several API endpoints for interaction with the inventory:

GET /health

Description: Checks if the inventory service and database are operational.

Possible Responses:

- **200 OK**

```
{
  "status": "OK",
  "db_status": "connected",
  "sample_item_check": {
    "id": 1,
    "name": "Sample Item",
    "category": "food",
    "price": 10.99,
    "description": "A test item",
    "count": 5
  }
}
```

- **ERROR** (e.g., 500 Internal Server Error)

```
{
  "status": "ERROR",
  "db_status": "disconnected",
  "error": "Database connection failed"
}
```

POST /api/v1/inventory/add

Description: Adds a new item to the inventory. The item must have a valid category, a positive price, and a non-negative count.

Request Body Example:

```
{
  "name": "Green T-Shirt",
  "category": "clothes",
  "price": 19.99,
  "description": "A comfortable green t-shirt",
  "count": 100
}
```

Possible Responses:

- **200 OK**

```
{
  "message": "Good added successfully"
}
```

- **422 Error: Unprocessable Entity** (e.g., if validation fails)

```
{
  "detail": [
    {
      "type": "greater_than_equal",
      "loc": [
        "body",
        "count"
      ],
      "msg": "Input should be greater than or equal to 0",
      "input": -1,
      "ctx": {
        "ge": 0
      }
    }
  ]
}
```

- **500 Internal Server Error** (e.g., database issues)

```
{
  "detail": "Failed to add good: Database insertion error"
}
```

PUT /api/v1/inventory/update/{good_id}

Description: Updates an existing inventory item using partial updates. Only provided fields will be updated.

Path Parameter:

- `good_id` (int): The ID of the item to be updated.

Request Body Example (partial update):

```
{
  "price": 24.99,
  "count": 120
}
```

Possible Responses:

- **200 OK**

```
{
  "message": "Good updated successfully"
}
```

- **404 Not Found** (if no item with the given ID exists)

```
{
  "detail": "Good not found"
}
```

- **422 Error: Unprocessable Entity** (if fields are invalid)

```
{
  "detail": [
    {
      "type": "greater_than_equal",
      "loc": [
        "body",
        "count"
      ],
      "msg": "Input should be greater than or equal to 0",
      "input": -1,
      "ctx": {
        "ge": 0
      }
    }
  ]
}
```

- **500 Internal Server Error** (database or unknown errors)

```
{
  "error": "Failed to update good: Internal database error"
}
```

GET /api/v1/inventory/{good_id}

Description: Retrieves an inventory item by its ID.

Path Parameter:

- `good_id` (int): The ID of the item to be retrieved.

Possible Responses:

- **200 OK**

```
{
  "id": 2,
  "name": "Green T-Shirt",
  "category": "clothes",
  "price": 19.99,
  "description": "A comfortable green t-shirt",
  "count": 100
}
```

- **404 Not Found** (if item does not exist)

```
{
  "detail": "Good not found"
}
```

- **500 Internal Server Error** (database or unknown errors)

```
{
  "error": "Database query error"
}
```

PUT /api/v1/inventory/deduct/{good_id}

Description: Deducts one unit from the inventory count of the specified item.

Path Parameter:

- `good_id` (int): The ID of the item to deduct.

Possible Responses:

- **200 OK**

```
{
  "message": "Stock deducted successfully"
}
```

- **404 Not Found** (if item does not exist)

```
{
  "detail": "Good not found"
}
```

- **422 Error: Unprocessable Entity** (if the item's count is already zero)

```
{
  "detail": [
    {
      "type": "greater_than_equal",
      "loc": [
        "body",
        "count"
      ],
      "msg": "Input should be greater than or equal to 0",
      "input": -1,
      "ctx": {
        "ge": 0
      }
    }
  ]
}
```

- **500 Internal Server Error** (database or unknown errors)

```
{
  "error": "Database query error"
}
```

3.2.3 Sales Service

The service exposes the following API endpoints:

GET /health

Description: Checks if the sales service and database are operational.

Possible Responses:

- 200 OK

```
{
  "status": "OK",
  "db_status": "connected",
  "records_found": 42
}
```

- 500 Internal Server Error (or other errors)

```
{
  "status": "ERROR",
  "db_status": "disconnected",
  "error": "Database connection failed"
}
```

GET /api/v1/sales/get

Description: Retrieves all purchase records from the sales database.

No Request Body.

Possible Responses:

- 200 OK

```
[
  {
    "id": 1,
    "good_id": 100,
    "customer_id": "johndoe",
    "amount_deducted": 19.99,
    "time": "2024-12-08T12:34:56Z"
  },
  {
    "id": 2,
    "good_id": 101,
    "customer_id": "janedoe",
    "amount_deducted": 29.99,
    "time": "2024-12-08T13:45:10Z"
  }
]
```

- 500 Internal Server Error

```
{
  "detail": "Internal server error"
}
```

POST /api/v1/sales/purchase/{customer_username}/{good_id}

Description: Processes the purchase of a specified good by a given customer. This endpoint communicates with the inventory and customer services to:

1. Verify the good exists and fetch its price.
2. Deduct the purchase price from the customer's wallet.
3. Deduct one unit of the good from the inventory.
4. Record the purchase in the sales database.

Path Parameters:

- **customer_username** (str): The username of the customer making the purchase.
- **good_id** (int): The ID of the good being purchased.

No Request Body.

Possible Responses:

- **200 OK**

```
{
  "message": "Purchase successful"
}
```

- **400 Bad Request** (e.g., good not found, customer not found, insufficient funds, or no stock)

```
{
  "detail": "Good with ID '999' not found"
}
```

Other examples for 400 Bad Request:

```
{
  "detail": "Customer 'johndoe' not found in the database"
}

{
  "detail": "Amount not enough"
}
```

```
{
  "detail": "Stock already zero"
}
```

- **500 Internal Server Error** (Any other unexpected issues)

```
{
  "detail": "Internal server error"
}
```

3.2.4 Reviews Service

The service exposes several API endpoints to manage reviews:

POST /api/v1/reviews/submit

Description: Submits a new review for an item. The review is initially set to a “needs_approval” status.

Request Body Example:

```
{
  "customer_id": "johndoe",
  "item_id": 123,
  "rating": 4,
  "comment": "Great product!"
}
```

Possible Responses:

- **200 OK**

```
{
  "message": "Posted johndoe's review for 123 successfully.
  Waiting for admin review to be completed ..."
}
```

- **400 Bad Request** (Invalid customer or item)

```
{
  "message": "Either customer 'johndoe' or
  item with id '123' doesn't exist."
}
```

- **409 Conflict** (Review already exists)


```
{
  "message": "johndoe's review for 123 already exists.
  If you want to update send a put request instead of post."
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try posting review again later",
  "errors": "Could not submit review"
}
```

PUT /api/v1/reviews/update

Description: Updates an existing review. Only provided fields are updated.

Request Body Example:

```
{
  "customer_id": "johndoe",
  "item_id": 123,
  "rating": 5,
  "comment": "Updated comment: I really love this item now!"
}
```

Possible Responses:

- **200 OK**

```
{
  "message": "Updated johndoe's review for 123 successfully.",
  "data": {
    "customer_id": "johndoe",
    "item_id": 123,
    "rating": 5,
    "comment": "Updated comment: I really love this item now!"
  }
}
```

- **202 Accepted** (No new data provided)

```
{
  "message": "johndoe's review for 123 update request processed,
  but no new data available"
}
```

- **404 Not Found** (No existing review found)

```
{
  "message": "johndoe's review for 123 not found"
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try updating again later",
  "errors": "Failed to update review"
}
```

DELETE /api/v1/reviews/delete?customer_id={customer_id}&item_id={item_id}

Description: Deletes an existing review for a given customer and item.

Query Parameters:

- **customer_id** (str): The ID of the customer who submitted the review.
- **item_id** (int): The ID of the item reviewed.

Possible Responses:

- **200 OK**

```
{
  "message": "Deleted review successfully."
}
```

- **404 Not Found** (Review not found)

```
{
  "message": "johndoe's review for 123 not found"
}
```

- **500 Internal Server Error**

```
{
  "message": "Internal Server Error. Please try deleting again later",
  "errors": "Failed to delete review"
}
```

GET /api/v1/reviews/item/{item_id}

Description: Retrieves all reviews for a given item.

Path Parameter:

- `item_id` (int): The ID of the item.

Possible Responses:

- **200 OK**

```
{
  "message": "Fetched reviews successfully for item_id 123.",
  "data": [
    {
      "customer_id": "johndoe",
      "item_id": 123,
      "rating": 4,
      "comment": "Great product!",
      "time": "2024-12-08T12:00:00Z",
      "flagged": "needs_approval"
    }
  ]
}
```

- **500 Internal Server Error** (e.g., database issues)

```
{
  "message": "Internal Server Error. Please try deleting again later",
  "errors": "Database Error"
}
```

GET /api/v1/reviews/customer/{customer_id}

Description: Retrieves all reviews submitted by a specific customer.

Path Parameter:

- `customer_id` (str): The customer's username.

Possible Responses: (Responses are similar to the item-based retrieval, but filtered by customer ID.)

GET /api/v1/reviews/details?customer_id={customer_id}&item_id={item_id}

Description: Retrieves a specific review by both customer ID and item ID.

Query Parameters:

- `customer_id` (str): The customer's username.
- `item_id` (int): The ID of the item.

Possible Responses: Similar to other GET endpoints, returns the single matching review or indicates not found or server error.

POST /api/v1/reviews/auth/login

Description: Authenticates a user using their username and password, returning a JWT token on success.

Request Body Example:

```
{
  "username": "johndoe",
  "password": "password123"
}
```

Possible Responses:

- **200 OK**

```
{
  "message": "Successfully logged in",
  "data": {
    "token": "JWT_TOKEN_HERE"
  }
}
```

- **404 Bad Request** (Invalid credentials)

```
{
  "detail": "Username or password is invalid"
}
```

- **500 Internal Server Error**

```
{
  "detail": "Server error message"
}
```

PUT /api/v1/reviews/moderate

Description: Moderates a review by changing its flagged status. Requires a valid Bearer token with appropriate permissions (e.g., admin).

Query Parameters:

- **customer_id** (str): The customer's username.
- **item_id** (int): The item ID.
- **new_flag** ("flagged"—"approved"—"needs_approval"): The new moderation status.

Headers:

- Authorization: Bearer JWT_TOKEN_HERE

Possible Responses:

- 200 OK

```
{
  "message": "Moderate review successfully for item_id 123
and customer_id johndoe. Changed it from
needs_approval to approved",
  "data": {
    "review": {
      "customer_id": "johndoe",
      "item_id": 123,
      "rating": 4,
      "comment": "Great product!",
      "time": "2024-12-08T12:00:00Z",
      "flagged": "approved"
    }
  }
}
```

- 400 Bad Request (Customer or item not found)

```
{
  "message": "Customer 'johndoe' or item with id '123' not found"
}
```

- 401 Unauthorized (Invalid token or insufficient permissions)

```
{
  "message": "Invalid Bearer Token"
}
```

- 404 Not Found (Review not found)

```
{
  "message": "johndoe's review for 123 not found"
}
```

- 500 Internal Server Error

```
{
  "message": "Internal Server Error. Please try moderating again later",
  "errors": "Database Error"
}
```

GET /health

Description: Checks if the review service and database are operational.

Possible Responses:

- 200 OK

```
{
  "status": "OK",
  "db_status": "connected"
}
```

- 500 Internal Server Error)

```
{
  "status": "ERROR",
  "db_status": "disconnected",
  "error": "Some error message"
}
```

3.3 Database Schema Design

The database schema is composed of five interconnected tables: `customer`, `wallet`, `inventory`, `review`, and `purchases`. Additionally, three ENUM types are defined to enforce data integrity. The schema is suitable for an e-commerce application with review functionality. The following sections detail each component of the design.

3.3.1 ENUM Types

- `marital_status_enum`: Defines possible values for a customer's marital status: 'single', 'married', 'divorced', 'widows'.
- `role_enum`: Specifies roles within the system: 'customer', 'moderator', 'admin'.
- `review_flagged_enum`: Represents the status of reviews: 'flagged', 'approved', 'needs_approval'.

3.3.2 Tables

3.3.3 Relationships

- **Customer-Wallet**: One-to-one relationship via `customer_id`.
- **Customer-Review**: One-to-many relationship via `customer_id`.
- **Inventory-Review**: One-to-many relationship via `item_id`.
- **Customer-Purchases**: One-to-many relationship via `customer_id`.
- **Inventory-Purchases**: One-to-many relationship via `good_id`.

Column Name	Description
name	Full name of the customer (TEXT).
username	Unique identifier for the customer (TEXT, PK).
password	Encrypted password (TEXT).
age	Age of the customer (INT, CHECK: age >= 0).
address	Residential address of the customer (TEXT).
gender	Gender of the customer (BOOLEAN).
marital_status	Customer's marital status (marital_status_enum).
role	Role of the customer in the system (role_enum).

Table 3.1: Customer Table

Column Name	Description
customer_id	Customer ID (TEXT, FK: customer(username), PK).
amount	Wallet balance (REAL, CHECK: amount >= 0).
last_updated	Timestamp of the last update (TIMESTAMP).

Table 3.2: Wallet Table

Column Name	Description
id	Unique identifier for the item (SERIAL, PK).
name	Name of the inventory item (TEXT, NOT NULL).
category	Category of the item (TEXT, NOT NULL).
price	Price of the item (REAL, NOT NULL).
description	Description of the item (TEXT, NOT NULL).
count	Available stock quantity (INTEGER, NOT NULL).

Table 3.3: Inventory Table

3.4 Database Schema Diagram

3.5 SQL Schema Definition

Below is the SQL code used to create the schema:

```
-- ENUM Types
CREATE TYPE marital_status_enum AS ENUM
    ('single', 'married', 'divorced', 'widows');
CREATE TYPE role_enum AS ENUM
    ('customer', 'moderator', 'admin');
```

Column Name	Description
customer_id	Customer ID (TEXT, FK: customer(username)).
item_id	Inventory item ID (INTEGER, FK: inventory(id)).
rating	Review rating (INT, CHECK: 1 <= rating <= 5).
comment	Text comment for the review (TEXT).
time	Timestamp of the review (TIMESTAMP).
flagged	Review status (review_flagged_enum).

Table 3.4: Review Table

Column Name	Description
id	Unique purchase ID (SERIAL, PK).
good_id	ID of the purchased inventory item (INTEGER, FK: inventory(id)).
customer_id	ID of the customer making the purchase (TEXT, FK: customer(username)).
amount_deducted	Amount deducted from the customer's wallet (REAL, NOT NULL).
time	Timestamp of the purchase (TIMESTAMP, NOT NULL).

Table 3.5: Purchases Table

```

CREATE TYPE review_flagged_enum AS ENUM
    ('flagged', 'approved', 'needs_approval');

-- Customer Table
CREATE TABLE customer (
    name TEXT,
    username TEXT PRIMARY KEY,
    password TEXT,
    age INT CHECK (age >= 0),
    address TEXT,
    gender BOOLEAN,
    marital_status marital_status_enum,
    role role_enum
);

-- Wallet Table
CREATE TABLE wallet (
    customer_id TEXT PRIMARY KEY REFERENCES customer(username),
    amount REAL CHECK (amount >= 0),
    last_updated TIMESTAMP
);

-- Inventory Table
CREATE TABLE inventory (

```

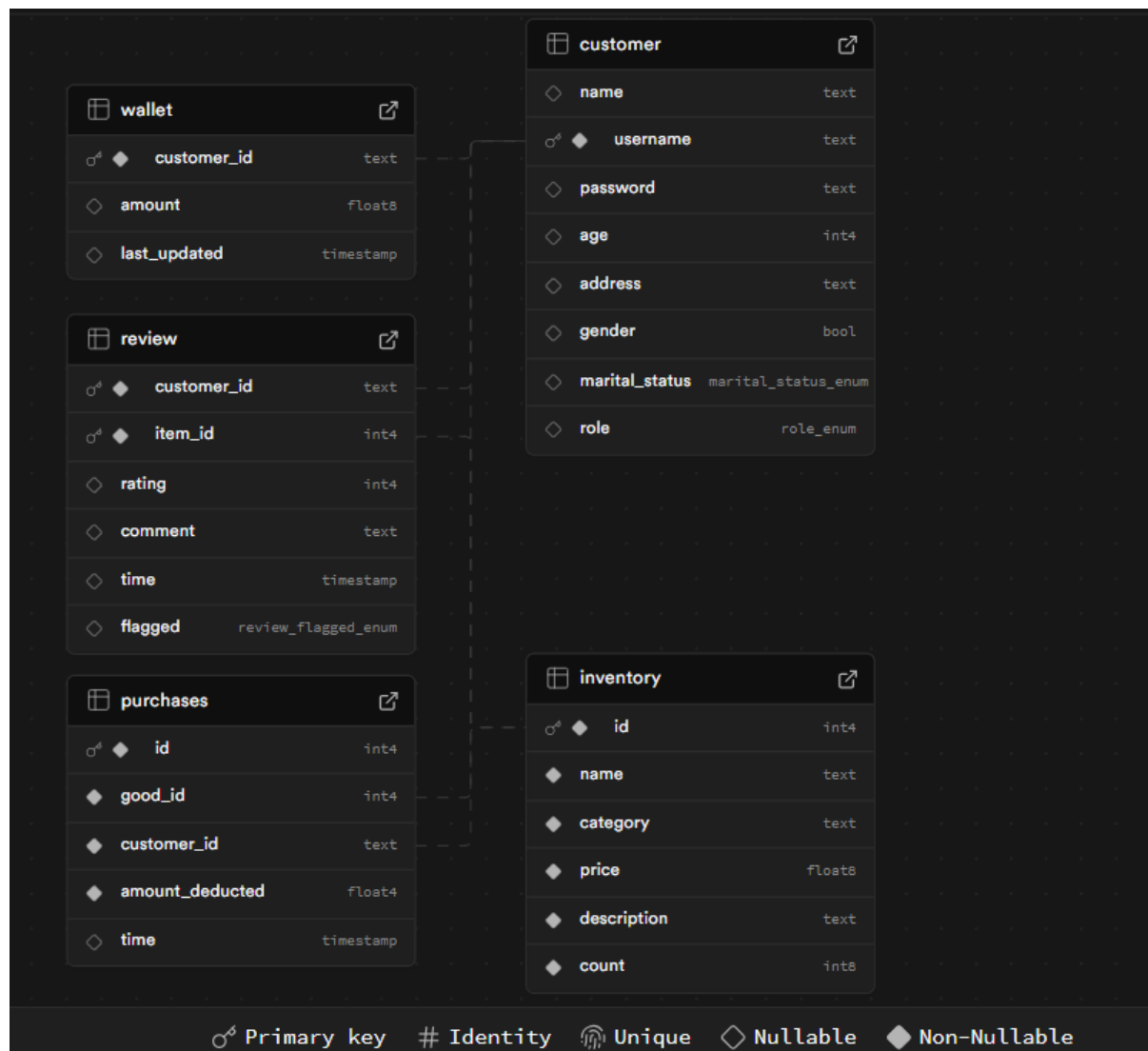



Figure 3.1: Database Schema Diagram

```

id SERIAL PRIMARY KEY,
name TEXT NOT NULL,
category TEXT NOT NULL,
price REAL NOT NULL,
description TEXT NOT NULL,
count INTEGER NOT NULL
);

-- Review Table
CREATE TABLE review (
  customer_id TEXT REFERENCES customer(username),
  item_id INTEGER REFERENCES inventory(id),
  rating INT CHECK (rating >= 0 AND rating <= 5),
  comment TEXT,
  time TIMESTAMP,
  flagged review_flagged_enum,

```

```
        PRIMARY KEY (customer_id, item_id)
    );

-- Purchases Table
CREATE TABLE purchases (
    id SERIAL PRIMARY KEY,
    good_id INTEGER NOT NULL REFERENCES inventory(id),
    customer_id TEXT NOT NULL REFERENCES customer(username),
    amount_deducted REAL NOT NULL,
    time TIMESTAMP NOT NULL
);
```

4. Error Handling and Validation

4.1 Error Management

The system adopts comprehensive error handling strategies at multiple levels:

- **Exception Handling:** All potential exceptions are caught and handled gracefully to prevent system crashes and provide meaningful feedback.
- **HTTP Status Codes:** Appropriate HTTP status codes are returned to the client to indicate the success or failure of a request, following RESTful API best practices.
- **Custom Error Messages:** Clear and informative error messages are provided to help clients understand the issue and take corrective action.
- **Logging:** Errors and exceptions are logged using the `loguru` library, aiding in debugging and monitoring.
- **Input Validation:** Pydantic models are used to validate incoming data, ensuring that only valid data is processed.

4.2 Validation

Pydantic Models All services utilize Pydantic's `BaseModel` for data validation. This ensures that:

- **Type Enforcement:** Data types are strictly enforced, preventing type-related errors.
- **Field Constraints:** Fields have constraints (e.g., non-negative integers) that are validated automatically.
- **Automatic Error Messages:** Validation errors generate informative messages without additional code.

5. Testing

5.1 Overview of Tools and Testing Strategy

We followed a structured testing strategy. The strategy includes multiple layers of testing: unit testing, integration testing, validation testing, and error handling. Below is a detailed description of the tools and methods used in the testing process.

5.2 Tools and Libraries

- **Pytest:** A versatile Python testing framework used for writing and automating unit tests, integration tests, and edge case validations.
- **Asyncio Support:** Testing of asynchronous FastAPI endpoints is achieved using `pytest-asyncio`.
- **FastAPI TestClient:** Provides a lightweight and efficient way to simulate HTTP requests to API endpoints and validate their responses.
- **Unittest.mock:** Facilitates mocking external dependencies, such as database operations, enabling isolated testing of individual components.
- **Pydantic Validation:** Ensures strict validation of input data and schema compliance using Pydantic's built-in validation mechanisms.
- **Supabase Mocking:** Custom mock implementations of the Supabase database interactions are used to simulate backend behavior.
- **Logging and Debugging:** Loguru is used for enhanced logging in test outputs to trace issues.

Testing Strategy

- **Unit Testing:** Focused on individual functions and methods to validate correctness. For example, database CRUD operations in models and API response behavior are extensively unit-tested.
- **Integration Testing:** Ensures smooth interactions between components like API endpoints and the database. Tests cover workflows such as customer registration, wallet updates, and error handling.
- **Validation Testing:** Validates inputs and schema definitions to prevent invalid data from entering the system. This includes testing boundary conditions for fields like age, wallet balance, and role attributes.

- **Error Handling:** Evaluates the system's ability to manage unexpected errors, such as database downtime or invalid API requests. Error responses are tested to ensure meaningful feedback to users.
- **Concurrency Testing:** For services handling high concurrency (e.g., wallet updates), tests simulate simultaneous requests to ensure consistency and correctness.

5.3 Customer Service

The Customer Service module tests are designed to validate the correctness, robustness, and scalability of functionalities related to customer operations. These tests are categorized as follows:

5.3.1 Unit Tests

Unit tests focus on individual components within the Customer Service module, ensuring that isolated methods and classes behave as expected. Key areas tested include:

- **API Endpoints:** Each API route is tested for expected behavior under normal and edge case scenarios, such as registering a customer, retrieving customer details, and deleting a customer.
- **Database Operations:** The `CustomerTable` class methods are validated for CRUD operations, including successful data insertion, updates, and retrieval from the database.
- **Validation Logic:** Input schemas are tested to enforce correct field constraints, such as non-negative ages and valid usernames.

5.3.2 Integration Tests

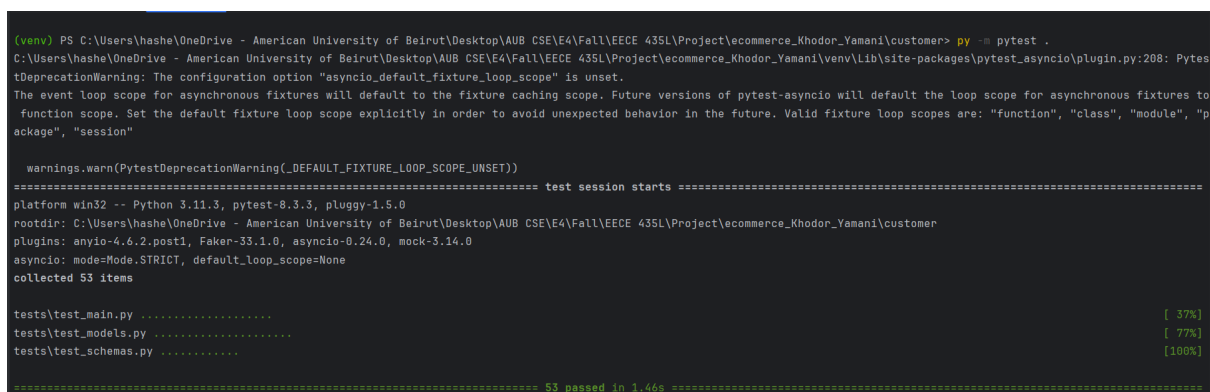
Integration tests validate interactions between different components of the Customer Service module. These tests ensure that:

- **API and Database Integration:** Endpoints correctly interact with the database, handling operations like creating a customer and linking them to a wallet.
- **Workflow Validation:** Complex operations, such as registering a customer followed by immediate retrieval, function as expected.
- **Error Propagation:** Failures in one layer (e.g., database unavailability) are accurately reflected in API responses.

5.3.3 Validation and Error Handling Tests

Validation tests ensure that inputs conform to required formats, and error handling tests validate the system's resilience to unexpected inputs or failures. This includes:

- **Invalid Data Inputs:** Testing with malformed or incomplete data to confirm appropriate error messages are returned.



```
(venv) PS C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer> py -m pytest .
C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\venv\Lib\site-packages\pytest_asyncio\plugin.py:208: PytestDeprecationWarning: The configuration option "asyncio_default_fixture_loop_scope" is unset.
The event loop scope for asynchronous fixtures will default to the fixture caching scope. Future versions of pytest-asyncio will default the loop scope for asynchronous fixtures to
function scope. Set the default fixture loop scope explicitly in order to avoid unexpected behavior in the future. Valid fixture loop scopes are: "function", "class", "module", "p
ackage", "session"

warnings.warn(PytestDeprecationWarning(_DEFAULT_FIXTURE_LOOP_SCOPE_UNSET))

===== test session starts =====
platform win32 -- Python 3.11.3, pytest-8.3.3, pluggy-1.5.0
rootdir: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer
plugins: anyio-4.6.2.post1, Faker-33.1.0, asyncio-0.24.0, mock-3.14.0
asyncio: mode=Mode.STRICT, default_loop_scope=None
collected 53 items

tests\test_main.py ..... [ 37%]
tests\test_models.py ..... [ 77%]
tests\test_schemas.py ..... [100%]

===== 53 passed in 1.46s =====
```

Figure 5.1: Output of pytest showing test results for the Customer module.

- **Error Responses:** Verifying meaningful error responses for scenarios such as attempting to delete a non-existent customer or accessing a customer with an invalid ID.
- **Concurrency Scenarios:** Simulating concurrent requests to update a customer's wallet to ensure consistency.

5.3.4 Edge Case Tests

Edge cases are tested to ensure the module behaves correctly under unusual conditions:

- Attempting to register a customer with an existing username.
- Retrieving details for a customer who does not exist in the database.
- Handling large payloads or high-frequency API requests without performance degradation.

5.4 Review Router

The Review Router module is designed to handle operations related to product reviews, including submission, updates, deletions, and moderation. To ensure its reliability and correctness, rigorous testing has been applied. This section describes the testing methodology and areas covered.

5.4.1 Unit Tests

Unit tests focus on verifying the functionality of individual components within the Review Router module. Key tests include:

- **API Endpoints:** Each endpoint, such as `/submit`, `/update`, and `/delete`, is tested for expected behavior under standard and edge case conditions.
- **Database Operations:** The `ReviewTable` class methods are validated for correct CRUD operations, including accurate handling of reviews and associated metadata.

- **Validation Logic:** Input schemas are tested to ensure compliance with rules, such as valid ratings (0-5), required fields, and correct data types.

5.4.2 Integration Tests

Integration tests validate the interaction between different components in the Review Router module. These tests cover:

- **API and Database Integration:** Endpoints correctly interact with the database to perform operations like submitting a new review or fetching reviews for a specific item.
- **Workflow Validation:** Multi-step workflows, such as submitting a review and retrieving it immediately, are tested to ensure data integrity.
- **Error Propagation:** Failures in one layer, such as database unavailability, are accurately reflected in API responses.

5.4.3 Validation and Error Handling Tests

Validation tests ensure that inputs meet expected formats, and error handling tests verify the system's resilience to unexpected inputs or failures. These include:

- **Invalid Input Data:** Testing with malformed data to ensure the system rejects it gracefully, such as invalid ratings or missing fields.
- **Error Responses:** Verifying that meaningful error messages are returned for scenarios like trying to delete a review that does not exist.

5.4.4 Edge Case Tests

Edge cases are tested to ensure the module behaves correctly under unusual conditions, such as:

- Submitting a review for a non-existent customer or product.
- Moderating a review when the user lacks administrative privileges.
- Fetching reviews when no reviews exist for a given product.

5.4.5 Moderation Workflow Tests

Specialized tests ensure that review moderation workflows are handled correctly:

- Verifying that only users with administrative roles can moderate reviews.
- Ensuring correct transitions between review states (`flagged`, `needs_approval`, `approved`).
- Testing the response to invalid tokens or unauthorized access attempts.

```

(venv) PS C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer> py -m pytest .
C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\venv\Lib\site-packages\pytest_asyncio\plugin.py:208: PytestDeprecationWarning: The configuration option "asyncio_default_fixture_loop_scope" is unset.
The event loop scope for asynchronous fixtures will default to the fixture caching scope. Future versions of pytest-asyncio will default the loop scope for asynchronous fixtures to
function scope. Set the default fixture loop scope explicitly in order to avoid unexpected behavior in the future. Valid fixture loop scopes are: "function", "class", "module", "p
ackage", "session"

  warnings.warn(PytestDeprecationWarning(_DEFAULT_FIXTURE_LOOP_SCOPE_UNSET))

===== test session starts =====
platform win32 -- Python 3.11.3, pytest-8.3.3, pluggy-1.5.0
rootdir: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer
plugins: anyio-4.6.2.post1, Faker-33.1.0, asyncio-0.24.0, mock-3.14.0
asyncio: mode=Mode.STRICT, default_loop_scope=None
collected 53 items

tests\test_main.py ..... [ 37%]
tests\test_models.py ..... [ 77%]
tests\test_schemas.py ..... [100%]

===== 53 passed in 1.46s =====

```

Figure 5.2: Output of pytest showing test results for the Review module.

5.5 Inventory Service

The Inventory Service module handles the management of goods, including adding, updating, retrieving, and deducting inventory. Rigorous tests have been conducted to ensure the reliability and correctness of these operations. This section outlines the testing methodology and key areas of focus.

5.5.1 Unit Tests

Unit tests validate the functionality of individual components within the Inventory Service module. Key areas tested include:

- **API Endpoints:** Endpoints such as `/add`, `/update`, `/deduct`, and `/get` are tested for expected behavior and correct error handling.
- **Database Operations:** The `InventoryTable` class methods are tested for CRUD operations, ensuring accurate insertion, updates, and retrieval of goods.
- **Validation Logic:** Models and schemas are tested to enforce constraints, such as positive prices, valid category names, and non-negative stock counts.

5.5.2 Integration Tests

Integration tests validate the interaction between the Inventory Service components. Key areas covered include:

- **Database and API Interaction:** Ensure that API endpoints correctly invoke database operations for tasks such as adding or updating goods.
- **Workflow Validation:** Verify that complex operations, such as deducting stock and returning updated counts, function correctly across multiple layers.
- **Error Propagation:** Confirm that database or service layer failures are accurately reflected in API error responses.

5.5.3 Validation and Error Handling Tests

These tests ensure robustness in handling invalid inputs and system failures:

- **Invalid Data:** Inputs with invalid fields, such as prices less than zero or overly long product descriptions, are tested to ensure proper rejection and meaningful error messages.
- **Error Responses:** Situations such as attempting to deduct stock when inventory is insufficient or trying to update a non-existent product are tested for correct error handling.
- **Concurrency Handling:** Simulate multiple requests to deduct inventory simultaneously to ensure stock consistency.

5.5.4 Edge Case Tests

Tests focus on ensuring correct behavior under unusual or unexpected conditions:

- Adding goods with minimal and maximal allowable field values.
- Deducting inventory to exactly zero to verify boundary behavior.
- Attempting updates on goods with invalid category names or excessive price values.

5.5.5 Business Logic Tests

Tests ensure that key business rules are upheld:

- Deducting inventory is disallowed when the stock count is zero.
- Updates on non-existent goods result in meaningful error messages.
- Goods added with missing optional fields default to appropriate values.

5.5.6 Performance and Scalability Tests

Performance tests verify that the system can handle high loads and large inventories efficiently:

- Simulating bulk additions of thousands of goods and ensuring database efficiency.
- High-frequency API calls for retrieval and updates, ensuring stability under load.

5.6 Sales Service

The Sales Service module is responsible for handling purchase-related operations, including fetching purchase records, recording sales, and managing inventory and wallet deductions. Comprehensive tests have been conducted to validate the accuracy, reliability, and error handling capabilities of this service. This section outlines the testing strategy and key areas covered.

```
(venv) PS C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\sales_service> py -m pytest .
C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\venv\Lib\site-packages\pytest_asyncio\plugin.py:208: PytestDeprecationWarning: The configuration option "asyncio_default_fixture_loop_scope" is unset.
The event loop scope for asynchronous fixtures will default to the fixture caching scope. Future versions of pytest-asyncio will default the loop scope for asynchronous fixtures to function scope. Set the default fixture loop scope explicitly in order to avoid unexpected behavior in the future. Valid fixture loop scopes are: "function", "class", "module", "package", "session"

warnings.warn(PytestDeprecationWarning(_DEFAULT_FIXTURE_LOOP_SCOPE_UNSET))

===== test session starts =====
platform win32 -- Python 3.11.3, pytest-8.3.3, pluggy-1.5.0
rootdir: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\sales_service
plugins: anyio-4.6.2.post1, Faker-33.1.0, asyncio-0.24.0, mock-3.14.0
asyncio: mode=Mode.STRICT, default_loop_scope=None
collected 35 items

tests\test_database.py .... [ 11%]
tests\test_main.py ..... [ 31%]
tests\test_models.py ..... [ 51%]
tests\test_service.py ..... [100%]

===== 35 passed in 0.98s =====
```

Figure 5.3: Output of pytest showing test results for the Inventory Service module.

5.6.1 Unit Tests

Unit tests validate individual components of the Sales Service module to ensure correctness in isolated scenarios. Key tests include:

- **Database Operations:** The `SalesTable` class methods are tested for correct behavior during operations such as recording a purchase and fetching purchase records.
- **Validation Logic:** Purchase models are validated to enforce constraints such as non-negative purchase amounts and mandatory fields like `good_id` and `customer_id`.
- **Service Functions:** Functions like `fetch_good_details`, `deduct_wallet_balance`, and `deduct_inventory` are tested to ensure accurate interaction with external services.

5.6.2 Integration Tests

Integration tests verify the seamless interaction between components within the Sales Service module. Key areas covered include:

- **Service and Database Interaction:** Tests ensure that service functions correctly call database operations, such as recording a sale after inventory and wallet updates.
- **External Service Dependencies:** Interaction with external services, such as the Inventory and Customer services, is validated for consistency and reliability.
- **Error Handling:** External service failures (e.g., insufficient wallet balance or inventory) are tested to ensure proper propagation of errors.

5.6.3 Validation and Error Handling Tests

These tests focus on the robustness of the Sales Service under various error conditions:

- **Invalid Input Data:** Scenarios such as negative purchase amounts or missing fields are tested to ensure appropriate error messages.

```
(venv) PS C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\sales_service> py -m pytest .
C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\venv\Lib\site-packages\pytest_asyncio\plugin.py:208: PytestDeprecationWarning: The configuration option "asyncio_default_fixture_loop_scope" is unset.
The event loop scope for asynchronous fixtures will default to the fixture caching scope. Future versions of pytest-asyncio will default the loop scope for asynchronous fixtures to
function scope. Set the default fixture loop scope explicitly in order to avoid unexpected behavior in the future. Valid fixture loop scopes are: "function", "class", "module", "package", "session"

warnings.warn(PytestDeprecationWarning(_DEFAULT_FIXTURE_LOOP_SCOPE_UNSET))
===== test session starts =====
platform win32 -- Python 3.11.3, pytest-8.3.3, pluggy-1.5.0
rootdir: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\sales_service
plugins: anyio-4.6.2.post1, Faker-33.1.0, asyncio-0.24.0, mock-3.14.0
asyncio: mode=Mode.STRICT, default_loop_scope=None
collected 35 items

tests\test_database.py .... [ 11%]
tests\test_main.py ..... [ 31%]
tests\test_models.py ..... [ 51%]
tests\test_service.py ..... [100%]

===== 35 passed in 0.98s =====
```

Figure 5.4: Output of pytest showing test results for the Sales Service module.

- **Service Failures:** Simulated failures in external services, such as database unavailability or invalid product IDs, are tested to ensure proper fallback mechanisms and error reporting.
- **Concurrency Handling:** Multiple concurrent purchase requests are simulated to ensure data integrity in inventory and wallet updates.

5.6.4 Edge Case Tests

Tests are designed to validate behavior in unusual scenarios:

- Attempting to purchase a good with zero stock or insufficient wallet balance.
- Fetching purchase records when none exist in the database.
- Processing a purchase for a non-existent customer or good.

5.6.5 Workflow Tests

Workflow tests validate multi-step operations in the Sales Service:

- **Purchase Processing:** Verifies that a purchase correctly deducts wallet balance, updates inventory, and records the sale.
- **Error Recovery:** Ensures that partial failures in a workflow (e.g., successful inventory deduction but failed wallet update) are properly handled and rolled back.

5.6.6 Performance and Scalability Tests

These tests assess the ability of the Sales Service to handle high transaction volumes:

- Simulating bulk purchases to ensure database efficiency and response time.
- High-frequency requests for fetching purchase records to test system scalability under load.

6. Deployment and Integration

6.1 Containerization of Services

Each microservice—Customers, Inventory, Sales, and Reviews—was containerized separately. This approach adheres to the microservices architecture principles, allowing services to be developed, tested, and deployed independently.

6.1.1 Dockerfiles for Each Service

For each service, a Dockerfile was created to define the container’s environment and the steps required to set up the service within the container. The key steps in each Dockerfile include:

- **Base Image Selection:** A lightweight Python image (e.g., `python:3.11-slim`) was used as the base to minimize the container size.
- **Setting the Working Directory:** The working directory inside the container was set to `/app`.
- **Dependency Installation:** The `requirements.txt` file was copied into the container, and all necessary Python dependencies were installed using `pip`.
- **Copying Application Code:** The application code was copied into the container.
- **Exposing Ports:** The appropriate port for each service was exposed (e.g., port 8000 for the Customers Service).
- **Running the Application:** The command to start the FastAPI application using Uvicorn was specified, binding the application to `0.0.0.0` to make it accessible outside the container.

6.1.2 Service-Specific Configurations

While the structure of the Dockerfiles is similar across services, each service specifies its unique port number and context:

- **Customers Service:** Runs on port 8000.
- **Reviews Service:** Runs on port 8001.
- **Sales Service:** Runs on port 8002.
- **Inventory Service:** Runs on port 8003.

6.1.3 Service Orchestration with Docker Compose

To manage and run all the services simultaneously, Docker Compose was used. The `docker-compose.yml` file defines the configuration for all services, enabling them to be started, stopped, and managed together.

`docker-compose.yml` Configuration

The Docker Compose configuration includes:

- **Services Definition:** Each service is defined with its build context, ports mapping, and environment variables.
- **Port Mapping:** Host ports are mapped to container ports to make the services accessible from the host machine.
- **Environment Variables:** Sensitive information such as the Supabase URL and key are passed as environment variables to ensure security and flexibility.

7. Documentation and Profiling

7.1 Documentation

Sphinx was chosen due to its ease of use and its ability to produce professional-looking documentation. The documentation includes:

- **API Documentation:** Detailed descriptions of all API endpoints, including parameters, responses, and examples.
- **Code Documentation:** Auto-generated documentation from docstrings in the code, providing insights into the internal workings of the services.

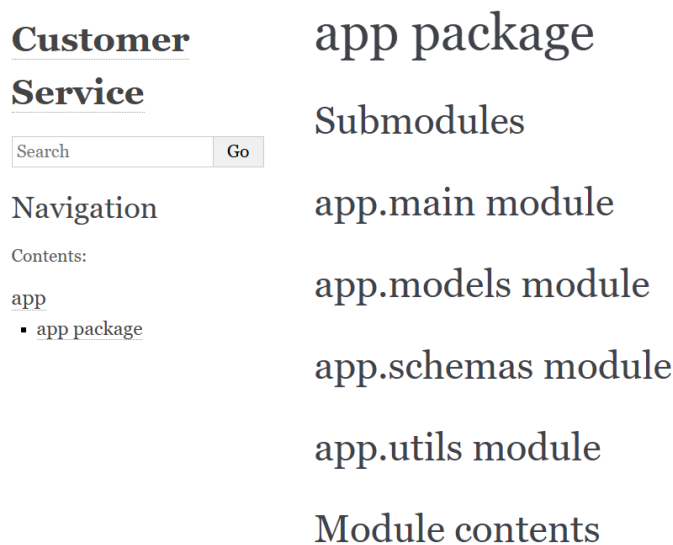


Figure 7.1: HTML Documentation

7.2 Performance Profiling

We conducted performance, memory, and code coverage profiling for the implemented services. The results for each aspect of profiling are included in the repository, under the file `route/profiler.txt`. Below are the profiling details:

- **Performance Profiling:** Execution time analysis was performed using `cProfile` and `LineProfiler` to identify bottlenecks in the code.
- **Memory Profiling:** Memory usage during execution was measured using `memory_profiler`.

- **Code Coverage Profiling:** Test coverage was assessed to ensure critical paths in the codebase are thoroughly tested.

The detailed profiling outputs and snapshots are available in the GitHub repository at `route/profiler.txt`.

```

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
  96                                     def mock_get_customer_sync():
  97          1    1321862.0    1e+06    100.0    asyncio.run(get_customer_profile())

Total time: 0.265679 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer\tests\api_profiler.py
Function: mock_charge_wallet_sync at line 100

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
 100                                     def mock_charge_wallet_sync():
 101          1    2656792.0    3e+06    100.0    asyncio.run(charge_wallet_profile())

Total time: 0.137855 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer\tests\api_profiler.py
Function: mock_deduct_wallet_sync at line 104

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
 104                                     def mock_deduct_wallet_sync():
 105          1    1378550.0    1e+06    100.0    asyncio.run(deduct_wallet_profile())

Profiling memory usage:
Filename: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer\tests\api_profiler.py

Line #      Mem usage  Increment  Occurrences  Line Contents
=====
 129         65.4 MiB    65.4 MiB          1  @profile
 130                                     async def memory_profile():
 131         65.4 MiB     0.0 MiB          1      await register_customer_profile()
 132         65.4 MiB     0.0 MiB          1      await delete_customer_profile()
 133         65.4 MiB     0.0 MiB          1      await update_customer_profile()
 134         65.4 MiB     0.0 MiB          1      await get_all_customers_profile()
 135         65.4 MiB     0.0 MiB          1      await get_customer_profile()
 136         65.4 MiB     0.0 MiB          1      await charge_wallet_profile()
 137         65.4 MiB     0.0 MiB          1      await deduct_wallet_profile()

```

Figure 7.2: Sample profiling output showing memory usage of customer service.

```

Profiling line-by-line:
Timer unit: 1e-07 s

Total time: 0.447642 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer\tests\api_profiler.py
Function: mock_register_customer_sync at line 80

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
  80                                     def mock_register_customer_sync():
  81          1    4476420.0    4e+06    100.0    asyncio.run(register_customer_profile())

Total time: 0.549675 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer\tests\api_profiler.py
Function: mock_delete_customer_sync at line 84

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
  84                                     def mock_delete_customer_sync():
  85          1    5496753.0    5e+06    100.0    asyncio.run(delete_customer_profile())

Total time: 0.278996 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\customer\tests\api_profiler.py
Function: mock_update_customer_sync at line 88

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
  88                                     def mock_update_customer_sync():
  89          1    2789964.0    3e+06    100.0    asyncio.run(update_customer_profile())

```

Figure 7.3: Sample profiling output showing execution time of customer service.

Name	Stmts	Miss	Cover	Missing
-----	-----	-----	-----	-----
__init__.py	0	0	100%	
app__init__.py	0	0	100%	
app\main.py	110	19	83%	130-132, 177, 186-188, 216, 229-231, 251, 265, 278-280, 310-312
app\models.py	111	26	77%	56-57, 98, 112, 138-140, 167-170, 224-226, 254, 258-290
app\schemas.py	130	20	85%	170, 215, 258, 297, 308-312, 358, 364-368, 420-423, 472, 476-479
tests\test_customer_api.py	0	0	100%	
tests\test_main.py	217	0	100%	
tests\test_models.py	261	1	99%	23
tests\test_schemas.py	52	0	100%	
tests\test_utils.py	0	0	100%	
-----	-----	-----	-----	-----
TOTAL	881	66	93%	

Figure 7.4: Customer service code coverage.

```

Profiling line-by-line:
Timer unit: 1e-07 s

Total time: 0.355859 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\inventory_service\tests\api_profiler.py
Function: mock_add_good_sync at line 52

Line #      Hits      Time  Per Hit  % Time  Line Contents
=====
52          1    3558589.0    4e+06   100.0      def mock_add_good_sync():
53          1    3558589.0    4e+06   100.0      asyncio.run(add_good_profile())

Total time: 0.641441 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\inventory_service\tests\api_profiler.py
Function: mock_update_good_sync at line 56

Line #      Hits      Time  Per Hit  % Time  Line Contents
=====
56          1    6414407.0    6e+06   100.0      def mock_update_good_sync():
57          1    6414407.0    6e+06   100.0      asyncio.run(update_good_profile())

Total time: 0.268133 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\inventory_service\tests\api_profiler.py
Function: mock_get_good_sync at line 60

Line #      Hits      Time  Per Hit  % Time  Line Contents
=====
60          1    2681328.0    3e+06   100.0      def mock_get_good_sync():
61          1    2681328.0    3e+06   100.0      asyncio.run(get_good_profile())

Total time: 0.5673 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\inventory_service\tests\api_profiler.py
Function: mock_deduct_good_sync at line 64

Line #      Hits      Time  Per Hit  % Time  Line Contents
=====
64          1    5673004.0    6e+06   100.0      def mock_deduct_good_sync():
65          1    5673004.0    6e+06   100.0      asyncio.run(deduct_good_profile())

```

Figure 7.5: Sample profiling output showing memory usage of inventory service.

```

Profiling memory usage:
Filename: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\inventory_service\tests\api_profiler.py

Line #      Mem usage  Increment  Occurrences  Line Contents
=====
83      54.5 MiB    54.5 MiB         1      @profile
84      54.5 MiB     0.0 MiB         1      async def memory_profile():
85      54.5 MiB     0.0 MiB         1      await add_good_profile()
86      54.5 MiB     0.0 MiB         1      await update_good_profile()
87      54.5 MiB     0.0 MiB         1      await get_good_profile()
88      54.5 MiB     0.0 MiB         1      await deduct_good_profile()

```

Figure 7.6: Sample profiling output showing execution time of inventory service.

Name	Stmts	Miss	Cover	Missing

__init__.py	0	0	100%	
app__init__.py	0	0	100%	
app\database.py	41	2	95%	71, 80
app\main.py	44	9	80%	26-28, 72-73, 92-93, 112-113
app\models.py	20	0	100%	
app\service.py	27	0	100%	
tests\test_database.py	117	0	100%	
tests\test_main.py	80	0	100%	
tests\test_schemas.py	49	0	100%	
tests\test_service.py	70	0	100%	

TOTAL	448	11	98%	

Figure 7.7: Inventory service code coverage.

Profiling memory usage:				
Filename: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\reviews\tests\api_profiler.py				
Line #	Mem usage	Increment	Occurrences	Line Contents
=====				
117	65.8 MiB	65.8 MiB	1	@profile
118				async def memory_profile():
119	65.8 MiB	0.0 MiB	1	await submit_review_profile()
120	65.8 MiB	0.0 MiB	1	await update_review_profile()
121	65.8 MiB	0.0 MiB	1	await delete_review_profile()
122	65.8 MiB	0.0 MiB	1	await get_item_review_profile()
123	65.8 MiB	0.0 MiB	1	await get_customer_review_profile()
124	65.8 MiB	0.0 MiB	1	await login_profile()

Figure 7.8: Sample profiling output showing memory usage of review service.

```

Profiling line-by-line:
Timer unit: 1e-07 s

Total time: 0.594015 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\reviews\tests\api_profiler.py
Function: mock_submit_review_sync at line 74

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
   74              1  5940146.0    6e+06   100.0      def mock_submit_review_sync():
   75              1  5940146.0    6e+06   100.0      asyncio.run(submit_review_profile())

Total time: 0.294102 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\reviews\tests\api_profiler.py
Function: mock_update_review_sync at line 78

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
   78              1  2941018.0    3e+06   100.0      def mock_update_review_sync():
   79              1  2941018.0    3e+06   100.0      asyncio.run(update_review_profile())

Total time: 0.29134 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\reviews\tests\api_profiler.py
Function: mock_delete_review_sync at line 82

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
   82              1  2913404.0    3e+06   100.0      def mock_delete_review_sync():
   83              1  2913404.0    3e+06   100.0      asyncio.run(delete_review_profile())

Total time: 0.161502 s
File: C:\Users\hashe\OneDrive - American University of Beirut\Desktop\AUB CSE\E4\Fall\EECE 435L\Project\ecommerce_Khodor_Yamani\reviews\tests\api_profiler.py
Function: mock_get_item_review_sync at line 86

Line #      Hits      Time  Per Hit   % Time  Line Contents
=====
   86              1  1615022.0    2e+06   100.0      def mock_get_item_review_sync():
   87              1  1615022.0    2e+06   100.0      asyncio.run(get_item_review_profile())

```

Figure 7.9: Sample profiling output showing execution time of review service.

Name	Stmts	Miss	Cover	Missing
__init__.py	0	0	100%	
app__init__.py	0	0	100%	
app\auth.py	48	27	44%	60-73, 85-92, 105-131, 141-158
app\main.py	119	16	87%	66, 105, 121-122, 156, 204-206, 278, 338, 374-376, 405-407
app\models.py	90	9	90%	62, 118-119, 223-239
app\schemas.py	137	21	85%	142, 186-187, 192-197, 253, 290, 296-300, 308, 358, 419, 489-494, 501, 509
tests\test_main.py	217	0	100%	
tests\test_models.py	130	0	100%	
tests\test_schemas.py	184	0	100%	
TOTAL	925	73	92%	

Figure 7.10: Review service code coverage.

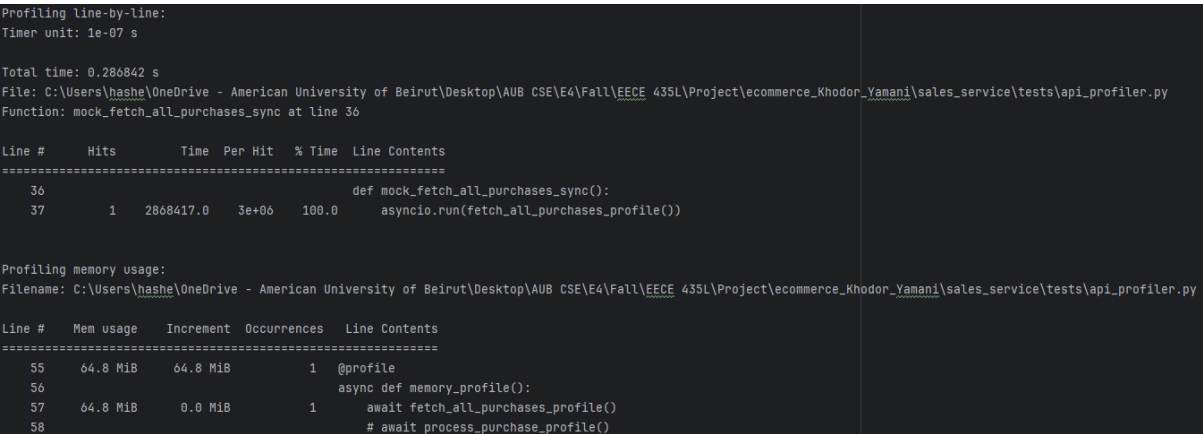


Figure 7.11: Sample profiling output showing memory usage and execution time of sales service.

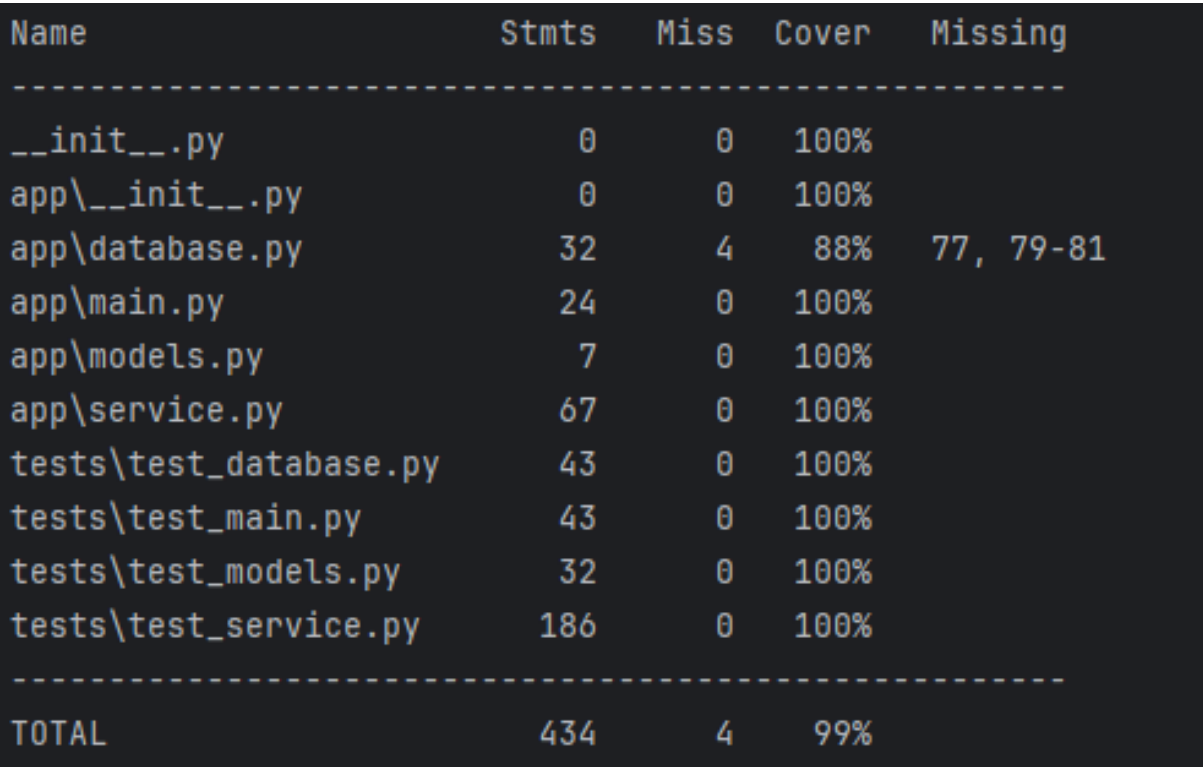


Figure 7.12: Sales service code coverage.

8. GitHub and Version Control

8.1 Repository Link

ecommerce_Khodor_Yamani Repository

Final changes on main branch.

9. Validation and Sanitization

Across all the services developed for this project (Customer, Inventory, Reviews, and Sales), we have adopted a consistent approach focused on using Pydantic models for input validation and leveraging parameterized queries or structured API calls to prevent injection attacks. Below is a high-level summary of these measures.

9.1 Use of Pydantic for Input Validation

At the core of our validation strategy is the use of Pydantic models. Each service defines request and response schemas using Pydantic. When FastAPI receives a request, it automatically:

- Checks the request body against the associated Pydantic schema.
- Ensures that all fields are of the correct type (e.g., `int`, `str`, `float`).
- Enforces any constraints specified (e.g., `rating` between 0 and 5, `price > 0`, maximum lengths for `comment` or `name`, and so forth).
- Rejects requests that do not meet these criteria with a `422 Validation Error` response, preventing invalid or malicious data from being processed further.

For example:

- In the **Customer Service**, schemas ensure that fields like `age` are non-negative and that `username` and `password` are strings of appropriate length.
- In the **Inventory Service**, schemas check that `price` is greater than zero and `count` is non-negative, preventing unrealistic or malformed product entries.
- In the **Reviews Service**, schemas ensure that `rating` is always within the acceptable range and that `comment` is a valid string. These strict validations prevent attempts to inject harmful input or submit nonsensical ratings.
- In the **Sales Service**, schemas verify that the `amount_deducted` for a purchase is non-negative, maintaining data integrity and preventing attempts to create purchases with invalid amounts.

This uniform approach ensures a baseline level of input safety throughout the system.

9.2 Parameterization and Structured Queries

To mitigate the risk of SQL injection or other injection-based attacks, none of the services directly concatenate user input into SQL strings. Instead, they rely on:

- **Parameterized Queries:** When interacting with the database, user inputs are passed as parameters to queries rather than directly embedded into SQL statements as strings. This ensures that user input cannot alter the structure of the queries.
- **Structured Client Calls:** The services utilize structured database clients (e.g., Supabase with PostgREST, or parameterized HTTP calls) that handle query composition internally. Because queries are formed through predefined methods like `.eq()`, `.insert()`, or `.update()`, user inputs are sanitized before they ever reach the database layer.

```
def update_wallet(self, user_id: str, amount: float) -> Optional[list[Wallet]]:
    """
    Updates the wallet balance for a customer.

    Args:
        user_id (str): The customer's ID.
        amount (float): The amount to add (positive) or deduct (negative).

    Returns:
        Optional[list[Wallet]]: The updated wallet details as a list, or None if an exception occurred.
    """

    try:
        wallet = self.get_wallet(user_id)
        if not wallet:
            return wallet

        updated_wallet = (
            self.client.table("wallet")
            .update({"amount": wallet[0].amount + amount})
            .eq(column="customer_id", user_id)
            .execute()
        )

        if updated_wallet.data:
            return [Wallet.model_validate(updated_wallet.data[0])]

        return []
    except Exception as e:
        logger.exception(e)
        return None
```

Figure 9.1: Example of Parametrized and Structured Query using supabase client

9.3 Sanitization and Data Integrity Checks

While Pydantic ensures that data is of the correct type and within acceptable ranges, we also consider:

- **Length Constraints:** Setting maximum lengths for text fields (e.g., product descriptions, review comments) prevents excessively long inputs from causing performance or storage issues.
- **Existence Checks:** Before performing actions like posting a review or processing a purchase, services verify that related entities (e.g., the customer, the inventory item) exist and are valid. This prevents dangling references and maintains referential integrity.
- **Descriptive Error Responses:** When validation fails, the services return explicit error messages. This helps clients understand what went wrong and encourages correct input formatting.

```
class Good(BaseModel):  
    """  
    Represents an inventory item.  
  
    :param name: The name of the item (maximum length: 100 characters).  
    :type name: str  
    :param category: The category of the item. Must be one of ['food', 'clothes', 'accessories', 'electronics'].  
    :type category: Literal["food", "clothes", "accessories", "electronics"]  
    :param price: The price of the item. Must be greater than 0.  
    :type price: float  
    :param description: A brief description of the item (maximum length: 255 characters).  
    :type description: str  
    :param count: The number of items available in stock. Must be non-negative.  
    :type count: int  
    """  
  
    name: str = Field(..., max_length=100)  
    category: Literal["food", "clothes", "accessories", "electronics"]  
    price: float = Field(..., gt=0)  
    description: str = Field(..., max_length=255)  
    count: int = Field(..., ge=0)
```

Figure 9.2: Example Pydantic model with field validations

10. User Authentication

10.1 Token-Based Authentication

The system employs token-based authentication to manage user sessions securely:

- **Authentication Tokens:** After successful login, the server generates an authentication token that encapsulates the user's identity and session information.
- **Token Storage:** The token is provided to the client, which stores it securely (e.g., in HTTP-only cookies or local storage).
- **Token Usage:** For each subsequent request that requires authentication, the client includes the token in the request headers.
- **Token Verification:** The server validates the token to authenticate the user before processing the request.

10.2 Authorization Management

10.2.1 Role-Based Access Control (RBAC)

The system utilizes Role-Based Access Control to manage user permissions. Different roles are assigned specific permissions that dictate what actions a user can perform:

- **Customer:** Regular users who can submit, update, and delete their own reviews.
- **Moderator:** Users with additional privileges to moderate reviews, including approving or rejecting reviews submitted by customers.
- **Admin:** Users with full access, capable of managing all reviews and user accounts.

11. Moderation

11.1 Moderation Features for Inappropriate Reviews

The system allows both admins and moderators to flag reviews. The moderation process includes the following features and functionalities:

11.1.1 Moderation Process Implementation

The moderation feature is designed using an API endpoint, `/moderate`, which enables authorized personnel to update the flagged status of reviews. The key components of this process are:

- **Endpoint Authentication:** Access to the `/moderate` endpoint is restricted to authenticated users with moderators or administrative privileges. Authentication is handled using token-based security particularly JWT.
- **Flagging Mechanism:** The endpoint allows for three flag statuses:
 - `flagged`: Indicates that the review has been flagged for inappropriate content.
 - `approved`: Confirms that the review adheres to content guidelines.
 - `needs_approval`: Marks the review for further review before publishing.
- **Validation Checks:** The system verifies that the review exists and is associated with the specified customer and item. If the review does not exist, a `404 Not Found` status is returned.
- **User Role Verification:** Only users with roles beyond "customer" (e.g., moderators or administrators) are allowed to change the flag status. Unauthorized users attempting to access this feature receive a `401 Unauthorized` response.
- **Database Integration:** The flagged status is updated in the database, and the updated review information is returned upon successful operation.

11.1.2 JWT Implementation for Authentication

To secure user authentication and authorization, we implemented JSON Web Tokens (JWTs) using the PyJWT library. Each JWT encapsulates user-specific claims and is cryptographically signed using the HS256 algorithm and a secret key. This ensures that tokens are tamper-proof and can be verified for integrity.

Token Payload

The payload of the JWT contains the following user-specific claims:

- **name**: The full name of the customer (e.g., "John Doe").
- **username**: The unique username of the customer (e.g., "johndoe").
- **age**: The customer's age as an integer (e.g., 30).
- **address**: The residential address of the customer (e.g., "123 Main Street").
- **gender**: The customer's gender represented as a boolean (**True** for male, **False** for female).
- **marital_status**: The marital status of the customer, such as "single" or "married".
- **role**: The role of the user in the system, such as "customer" or "admin".
- **exp**: The expiration time of the token, defined as a UTC timestamp to limit its validity.

Token Lifecycle

The tokens are created using the `create_access_token` function, which takes a `Customer` object and an optional expiration delta. By default, tokens expire after one hour. The expiration claim (**exp**) ensures tokens are automatically invalidated after this period.

Tokens are verified using the `decode_access_token` function. This function validates the signature and checks for expiration. If the token is invalid or expired, appropriate error messages are returned.

Security Measures

To enhance security:

- Tokens are signed with a secret key using the HS256 algorithm.
- The payload excludes sensitive data, such as passwords.
- Token expiration times reduce exposure to token misuse.

12. Additional Professional Tasks

This section details the implementation of various system components, including JWT-based authentication, logging, asynchronous support, database indexing, and cloud-based hosting solutions.

12.1 JWT-Based Authentication

JWT (JSON Web Token) is utilized for secure authentication and authorization. The implementation uses the PyJWT library to create and verify tokens, ensuring role-based access control (RBAC) as discussed in 11.1.2.

12.2 Logging and Error Monitoring

The `loguru` library is integrated for efficient logging and error monitoring. This enables:

- Real-time logging of API requests, responses, and errors for debugging.
- Contextual error tracing with stack traces for asynchronous operations.
- Centralized logging, which can be redirected to external monitoring tools for further analysis.

```
2024-12-08 22:27:48.858 | INFO | app.models:get_reviews:115 - Fetching all reviews with filters: {'item_id': 1, 'customer_id': 'testuser'}
2024-12-08 22:27:48.862 | SUCCESS | app.models:get_reviews:118 - No reviews are available
2024-12-08 22:27:48.863 | INFO | app.models:submit_review:81 - Verifying that item with id '1' and customer with id 'testuser' exist
2024-12-08 22:27:48.865 | INFO | app.models:submit_review:84 - Submitting review: {'customer_id': 'testuser', 'item_id': 1, 'rating': 5, 'comment': 'Excellent product!', 'time': None, 'flagged': 'needs_approval'}
2024-12-08 22:27:48.868 | INFO | app.models:submit_review:87 - {'customer_id': 'testuser', 'item_id': 1, 'rating': 5, 'comment': 'Excellent product!', 'flagged': 'needs_approval'}
2024-12-08 22:27:48.875 | SUCCESS | app.models:submit_review:91 - Successfully submitted review
2024-12-08 22:27:49.237 | INFO | app.models:get_reviews:113 - Fetching all reviews with filters: {'item_id': 1, 'customer_id': 'testuser'}
2024-12-08 22:27:49.385 | SUCCESS | app.models:get_reviews:118 - Successfully fetched the reviews
2024-12-08 22:27:49.386 | INFO | app.models:update_review:136 - Updating review 1,testuser: {'customer_id': 'testuser', 'item_id': 1, 'rating': 4, 'comment': 'Updated my review: great product!', 'time': '2024-12-08T20:27:53.285449', 'flagged': 'needs_approval'}
2024-12-08 22:27:49.521 | SUCCESS | app.models:update_review:146 - Successfully updated the review
2024-12-08 22:27:49.591 | INFO | app.models:get_reviews:115 - Fetching all reviews with filters: {'item_id': 1, 'customer_id': 'testuser'}
2024-12-08 22:27:49.744 | SUCCESS | app.models:get_reviews:118 - Successfully fetched the reviews
2024-12-08 22:27:49.744 | INFO | app.models:delete_review:209 - Deleting testuser's review of item 1
2024-12-08 22:27:49.893 | SUCCESS | app.models:delete_review:213 - Successfully delete the review
2024-12-08 22:27:50.004 | INFO | app.models:get_reviews:113 - Fetching all reviews with filters: {'item_id': 1}
2024-12-08 22:27:50.181 | SUCCESS | app.models:get_reviews:118 - Successfully fetched the reviews
2024-12-08 22:27:50.258 | INFO | app.models:get_reviews:113 - Fetching all reviews with filters: {'customer_id': 'testuser'}
2024-12-08 22:27:50.420 | SUCCESS | app.models:get_reviews:118 - No reviews are available
2024-12-08 22:27:51.118 | INFO | app.models:get_reviews:113 - Fetching all reviews with filters: {'item_id': 1, 'customer_id': 'testuser'}
2024-12-08 22:27:51.260 | SUCCESS | app.models:get_reviews:118 - No reviews are available
2024-12-08 22:27:51.260 | INFO | app.models:submit_review:81 - Verifying that item with id '1' and customer with id 'testuser' exist
2024-12-08 22:27:51.261 | INFO | app.models:submit_review:84 - Submitting review: {'customer_id': 'testuser', 'item_id': 1, 'rating': 5, 'comment': 'Excellent product!', 'time': None, 'flagged': 'needs_approval'}
2024-12-08 22:27:51.263 | INFO | app.models:submit_review:87 - {'customer_id': 'testuser', 'item_id': 1, 'rating': 5, 'comment': 'Excellent product!', 'flagged': 'needs_approval'}
```

Figure 12.1: Sample logging output showing execution details with timestamp.

12.3 Asynchronous Operations

FastAPI powers the application, leveraging Python's asynchronous capabilities. Key benefits include:

- Non-blocking I/O ensures efficient handling of concurrent requests.
- Improved performance for I/O-intensive tasks such as database queries and network calls.
- Simplified code structure for async database operations and token management.

12.4 Cloud-Hosted Database with Indexing

The database is hosted on a cloud platform using **Supabase**, ensuring scalability and reliability. To optimize query performance, strategic indexing is applied:

- **Customer Table:**

- Primary key: `customer.username` (already indexed).
- Index for frequent filtering by role:

```
CREATE INDEX idx_customer_role ON customer (role);
```

- Index for marital status filtering:

```
CREATE INDEX idx_customer_marital_status ON  
customer (marital_status);
```

- **Wallet Table:**

- Primary key: `customer_id`, referencing `customer.username`.

- **Inventory Table:**

- Index for filtering by category:

```
CREATE INDEX idx_inventory_category ON inventory (category);
```

- Index for name-based searches:

```
CREATE INDEX idx_inventory_name ON inventory (name);
```

- **Review Table:**

- Primary key: (`customer_id`, `item_id`) (already indexed).
- Index for filtering by flagged status:

```
CREATE INDEX idx_review_flagged ON Review (flagged);
```

- **Purchases Table:**

- Index for frequent filtering by `customer_id`:

```
CREATE INDEX idx_purchases_customer_id  
ON purchases (customer_id);
```

- Index for sorting/filtering by purchase time:

```
CREATE INDEX idx_purchases_time ON purchases (time);
```

- Index for joins or filters by good_id:

```
CREATE INDEX idx_purchases_good_id ON purchases (good_id);
```

12.5 Health Checks

Health-check APIs are implemented for monitoring service availability and performance. These APIs periodically verify:

- Database connectivity.
- Service responsiveness.
- Token validation and authentication mechanisms.

References

-

A. Appendix A: Supporting Materials



Figure A.1: Customer Charge Wallet - 200 Response



Figure A.2: Customer Charge Wallet - 404 Response



Figure A.3: Customer Deduct - 200 Response

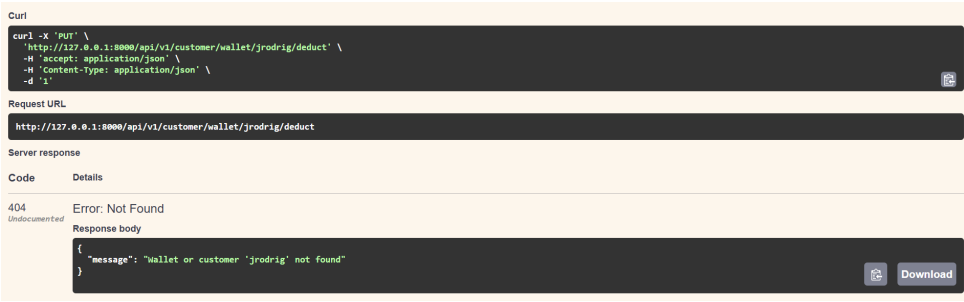


Figure A.4: Customer Deduct - 404 Response

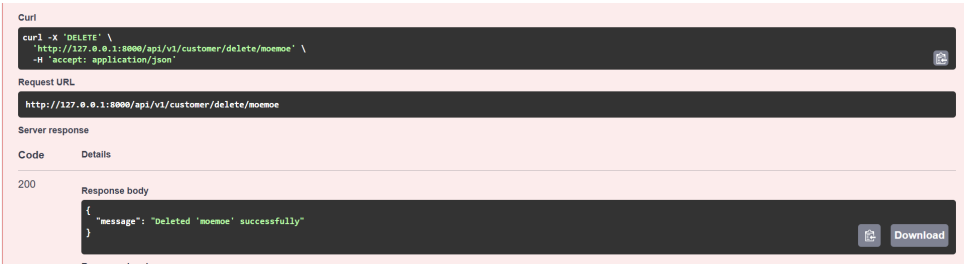


Figure A.5: Customer Delete - 200 Response

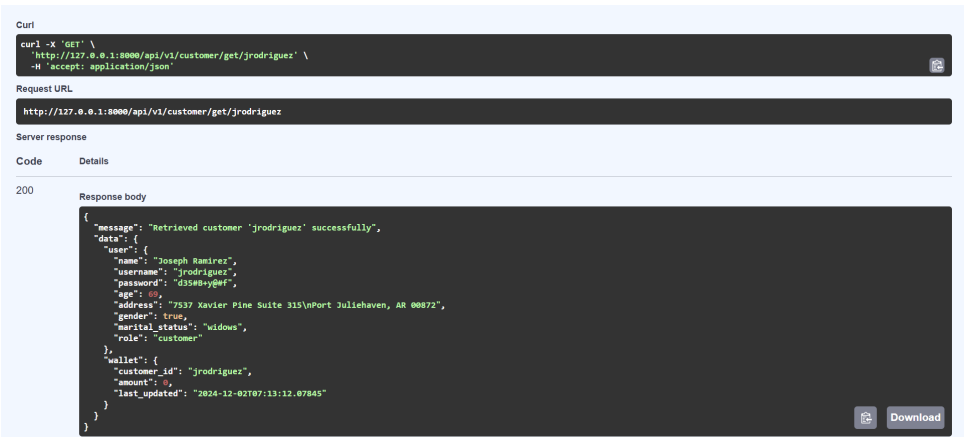


Figure A.6: Customer Get - 200 Response



Figure A.7: Customer Get All Response



Figure A.8: Customer Register - 200 Response



Figure A.9: Customer Register - 404 Response



Figure A.10: Customer Register - 409 Response

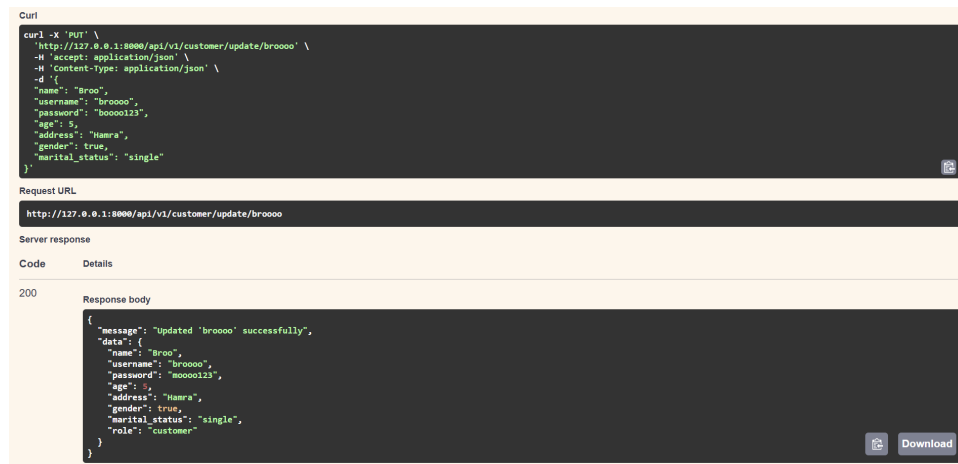


Figure A.11: Customer Update - 200 Response



Figure A.12: Customer Update - 404 Response



Figure A.13: Good Add - 200 Response

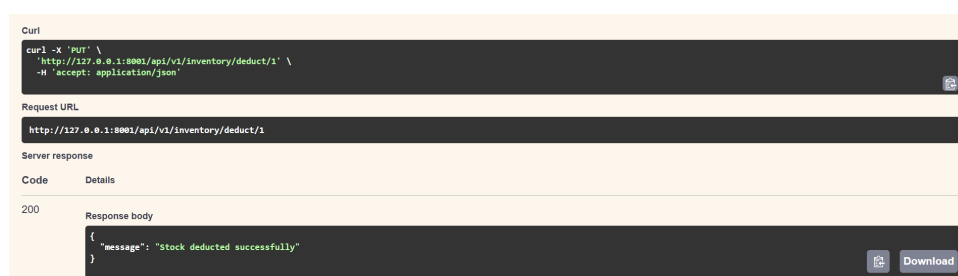


Figure A.14: Good Deduct - 200 Response

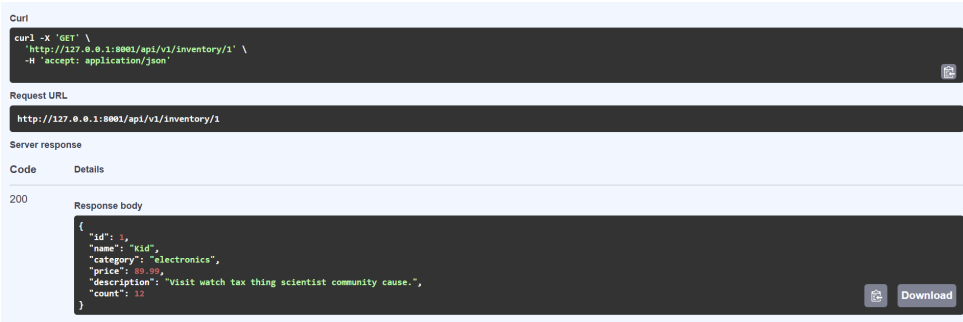


Figure A.15: Good Get - 200 Response



Figure A.16: Good Update - 200 Response

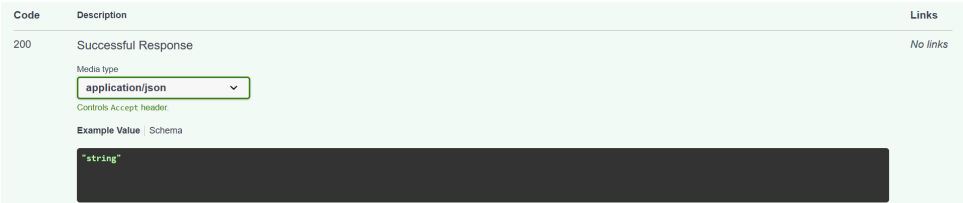


Figure A.17: Purchase - 200 Response (1)



Figure A.18: Purchase - 200 Response



Figure A.19: Sales Get - 200 Response

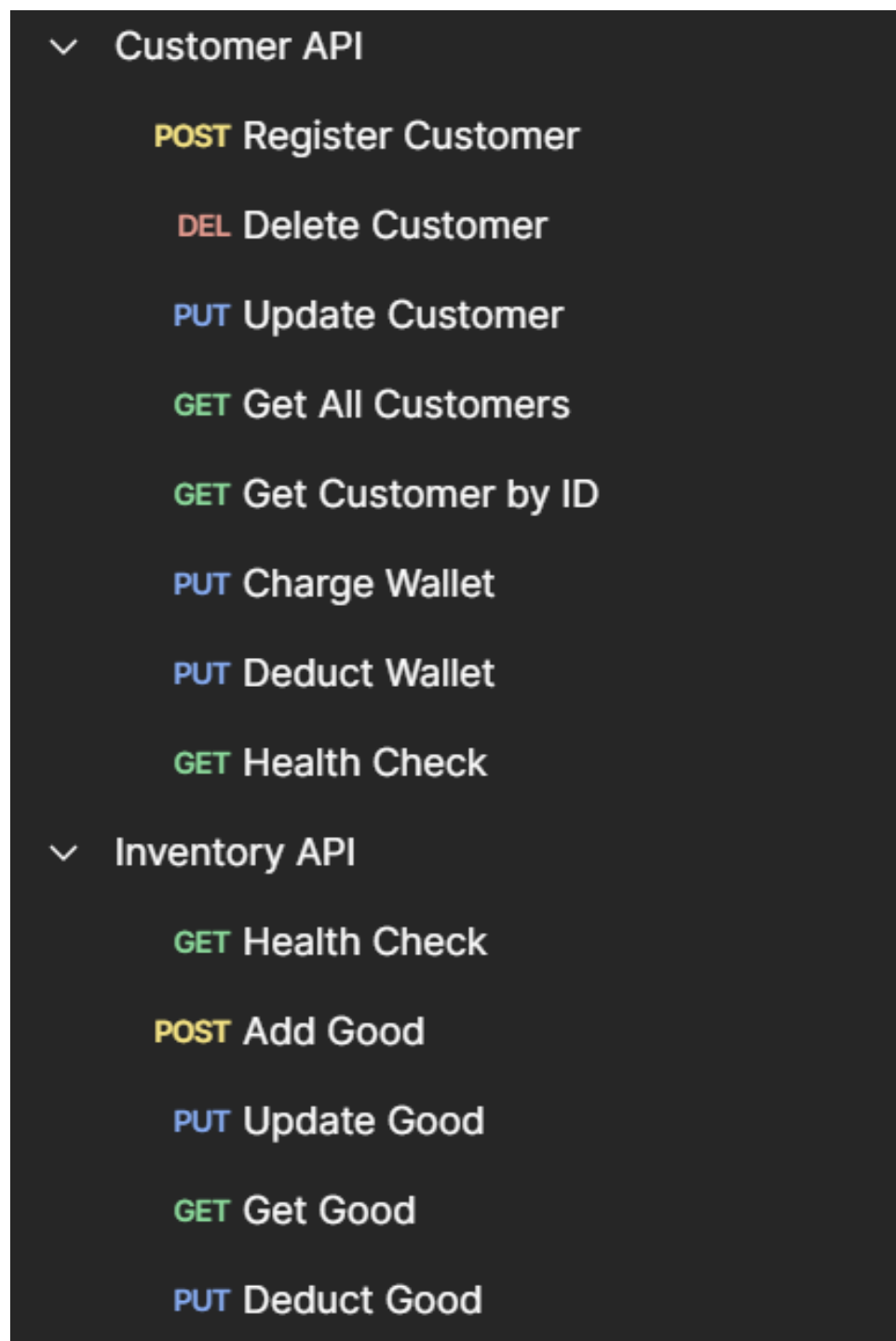


Figure A.20: Postman - Test 1

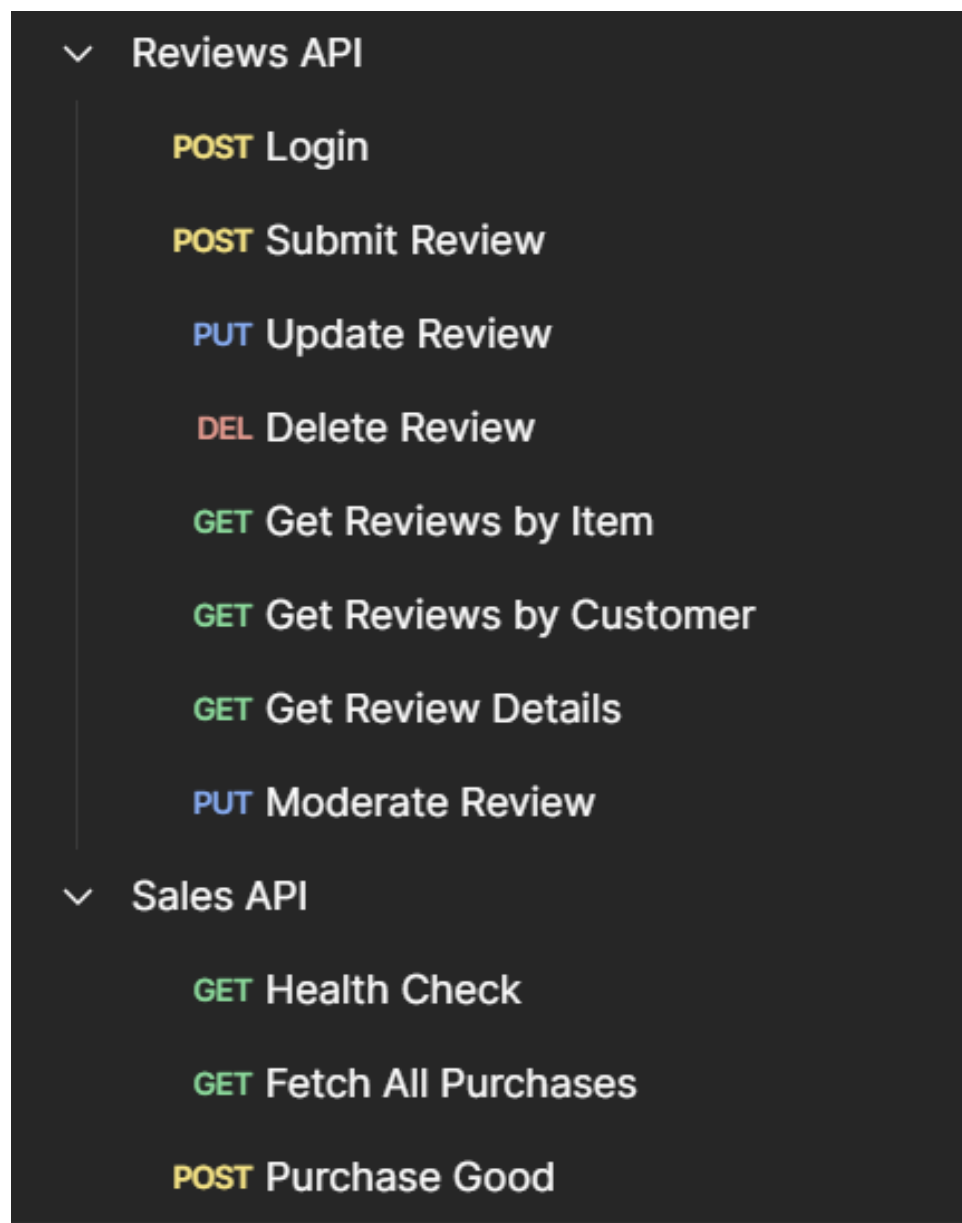


Figure A.21: Postman - Test 2

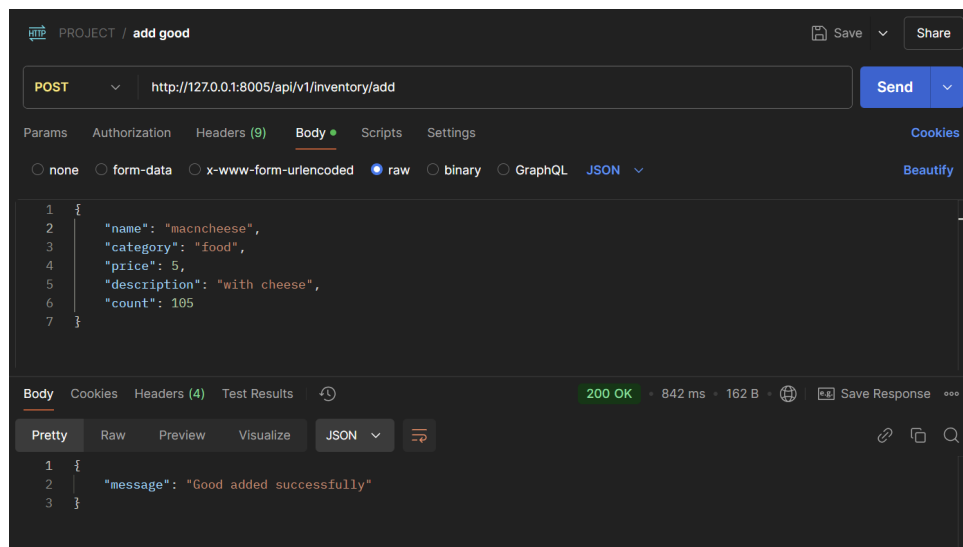


Figure A.22: Postman - Final Test